Deep Q learning

Description

As described in README.md, in this project we solve a navigation problem where an agent has to explore a large square world and try to get as many yellow bananas as possible while avoiding the blue ones.

In order to solve this problem, we have selected the well-known Deep Q learning approach. It is similar to the tabular Q-learning algorithm, but instead of using a lookup table or dictionary to store each state action pair values, it uses a function approximator to find the underlying relationships between the input (state) and the output (action). In this case, a nonlinear function approximator is used, a neural network.

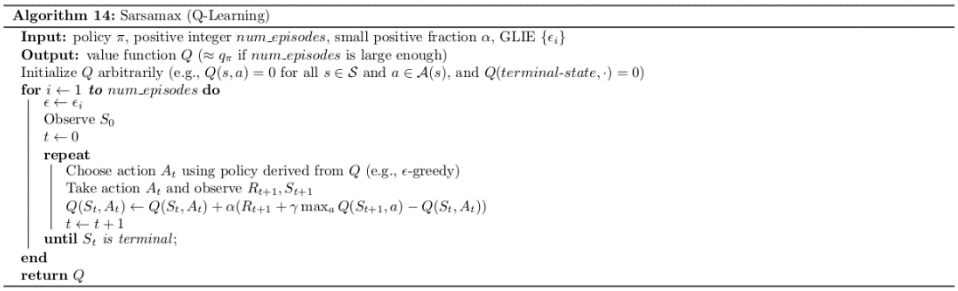


Figure 1. Q-learning

In our environment the input data are originally images; thus, the neural network is fed with raw pixels. However, such a large input space could not be processes correctly. In order to fix that, a CNN is used to reduce the dimensionality and get a significant feature vector of 37 elements, which resume the information of agent's velocity and its perception of the objects that surround it in its forward direction. All this initial processing is not covered in the current project and it's done by the environment, although it could be done manually if wanted.

Reinforcement learning is known to be unstable when used with a nonlinear function approximator such as neural networks. Hence, these instabilities have been addressed with **experience replay** and **target network** as explained in [1]. These mechanisms are used to minimize the correlation between:

- The sampled experiences
- The target (calculated in each visit inside the episodes) and the action-value parameters (weights of the neural network) that are constantly updating.

Furthermore, DQN tends to overestimate action values. Thus, a very popular approach to overcome this issue is to use **Double Q-learning,** which will be covered in this project.

Moreover, during experimentation has been found out that the C parameter (see in the section of Hyperparameters if more info is needed) is very sensitive to value changes. Because of that, I have considered interesting to get the results for different values.

To sum up, this project will resolve the explained navigation problem, analyzing the results between DQN, Double DQN and different C values. Moreover, for the interest of the reader, the provided code is also possible to use to evaluate the trained models using the model weights in each case.

Learning algorithm

The DQN has been implemented as described in [1], as shown in the below pseudocode:

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

Figure 2. DQN

On the other hand, Double DQN follows the same type of implementation, but the main difference could be found in the target calculation.

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^{-})$$

Instead of doing it directly with the target network, we decouple the action selection to avoid a possible overestimation. To do that, we rewrite the target calculation as:

$$Y_t^{\text{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

Double Deep Q-learning [2] uses the current/local network to select the best action in the next state, and the target network to obtain the q-value related to that action. Hence, the local network is used only to select the action and the target to evaluate it.

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t')$$

Hyperparameters

The criteria to select hyperparameters value may change from problem to problem. As reference, the values selected in [1] have been considered. Referring to the addressed problem, the main hyperparameters values have been set in the following way:

parameter	value	description
BUFFER_SIZE	100,000	Updates into the model are sample from this number of most recent experience tuples
BATCH_SIZE	32	Number of training cases over which each gradient update is computed
GAMMA	0.99	Discount factor used for the Q-learning update
C	{1250,5000}	Number of steps that have to be carried between target network updates
TAU	0.001	Soft update required parameter that set how the target network is updated respect the local network
LR	0.00005	Learning rate used for the optimizer

In the literature it is also considered a soft update version to update the target network parameters. In that version, the target network would be updated more frequently although a tau variable would be needed to set the way in which the update is done. The code has been developed to be able to choose both type of updates and both approaches have been explored for the resolution of this project, although the attached results are for the classical/hard update.

Moreover, other training parameters have been selected as shown below:

parameter	value	description
n_episodes	1500	Number of episodes that the agent is going to be executed
eps_start,eps_end_eps_decay	{1.0, 0.01, 0.995}	Epsilon related configuration values

Neural Network

It has been set a **sequential** schema with an architecture composed of:

- An input linear layer of 64 neurons
- A hidden linear layer of 64 neurons
- An output linear layer 4 neurons (equal to the number of actions available)

where ReLu activation functions are used in the input and hidden layers.

Moreover, Adam has been selected as optimizer and the learning rate has been *conservative* fixing a value of 0.0005 from the beginning.

Plot of rewards

As explained in the introduction, we have trained 4 different agents:

- DQN
  - With C =1250
  - With C = 5000
- Double DQN (DDQN)
  - With C =1250
  - With C = 5000

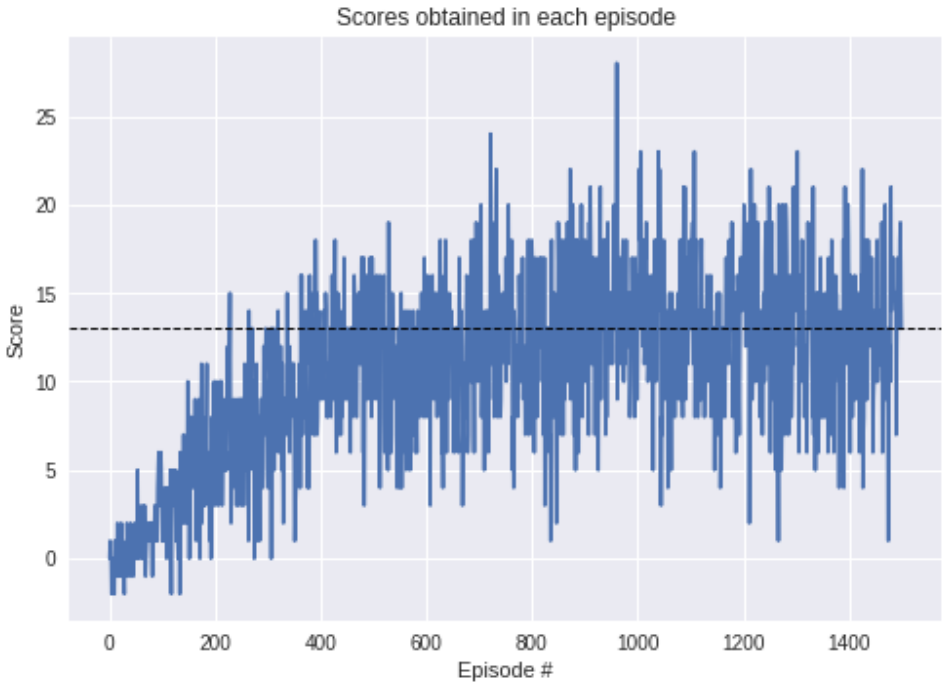
The problem is considered solved when the agent is able to average a score of +13 over 100 consecutive episodes.

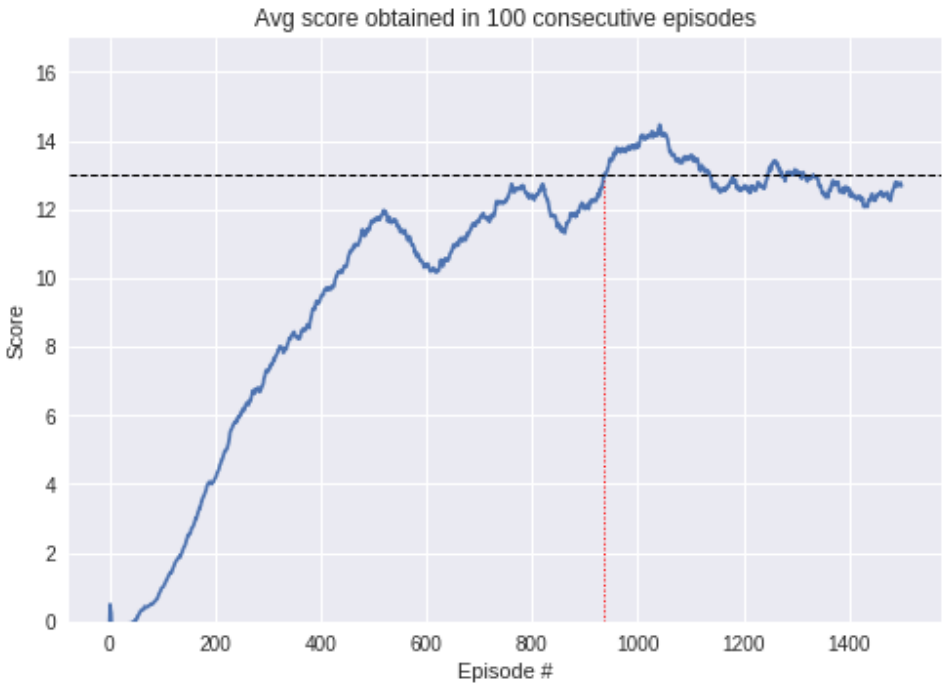
We provide the raw scores obtained for each agent during the training phase alongside the checkpoint in which the environment has been solved. Additionally, we provide another graph where it could be seen the average score obtained through 100 consecutive episodes for each episode. Finally, the average score and standard deviation obtained after the environment has been solved are also analysed in order to see the behaviour and the trend in the learning process.

DQN_c1250

Problem solved in 937episodes. After being solved, each episode has:

- Avg score: 13.01
- std: 3.93

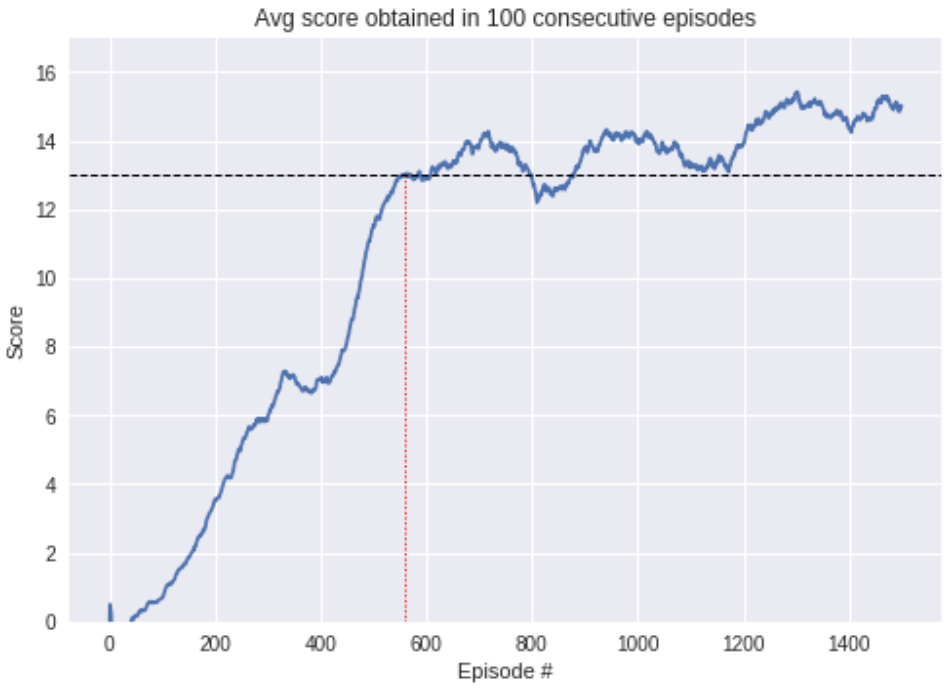




DQN_c5000

Problem solved in 557 episodes. After being solved, each episode has:

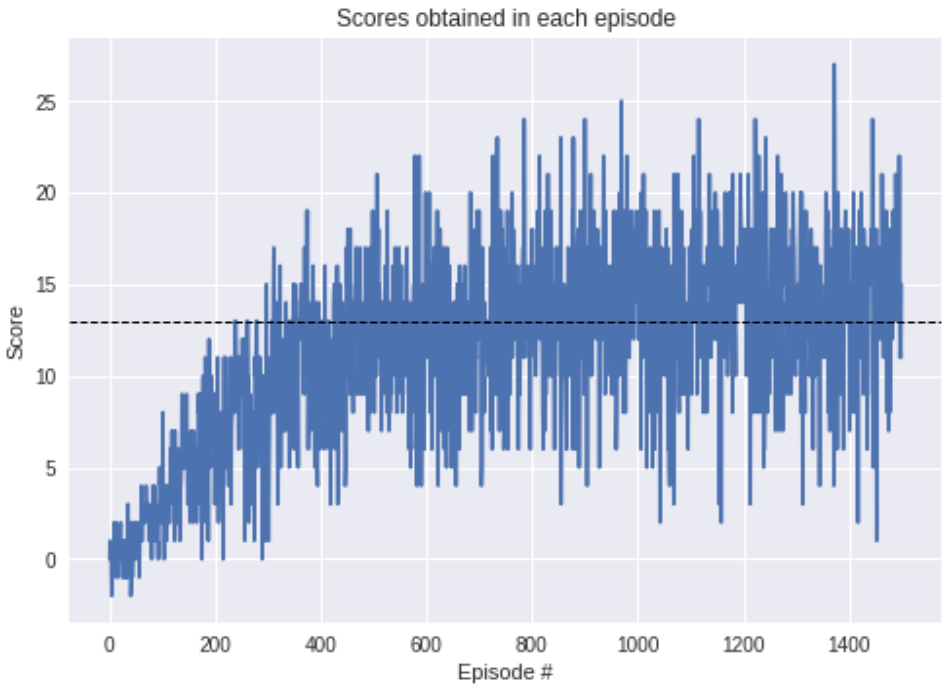
- Avg score: 13.99
- std: 4.07

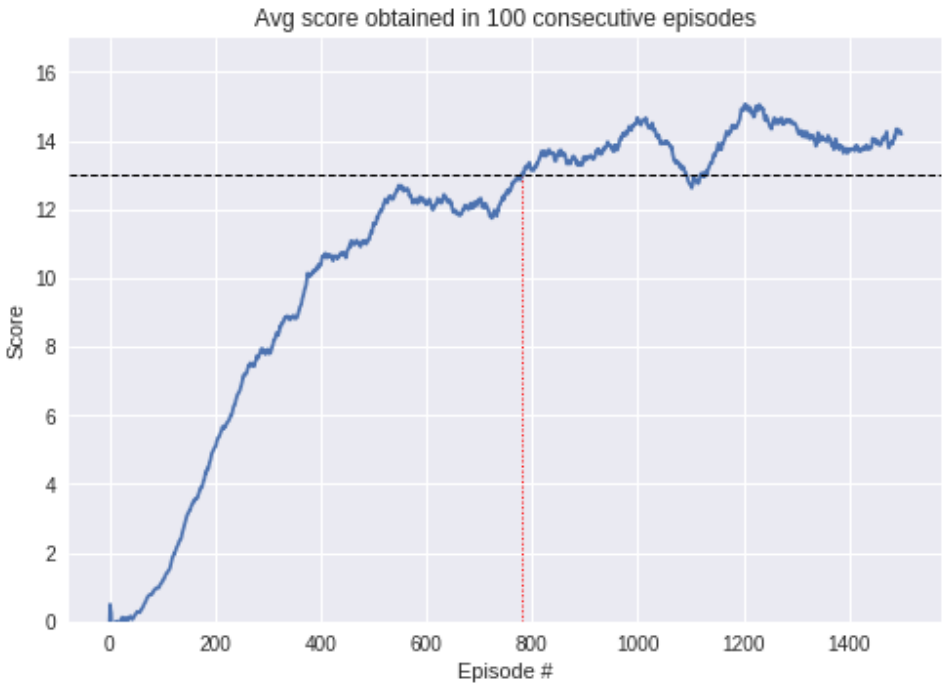


DDQN_c1250

Problem solved in 782 episodes. After being solved, each episode has:

- Avg score: 14.0
- std: 4.08

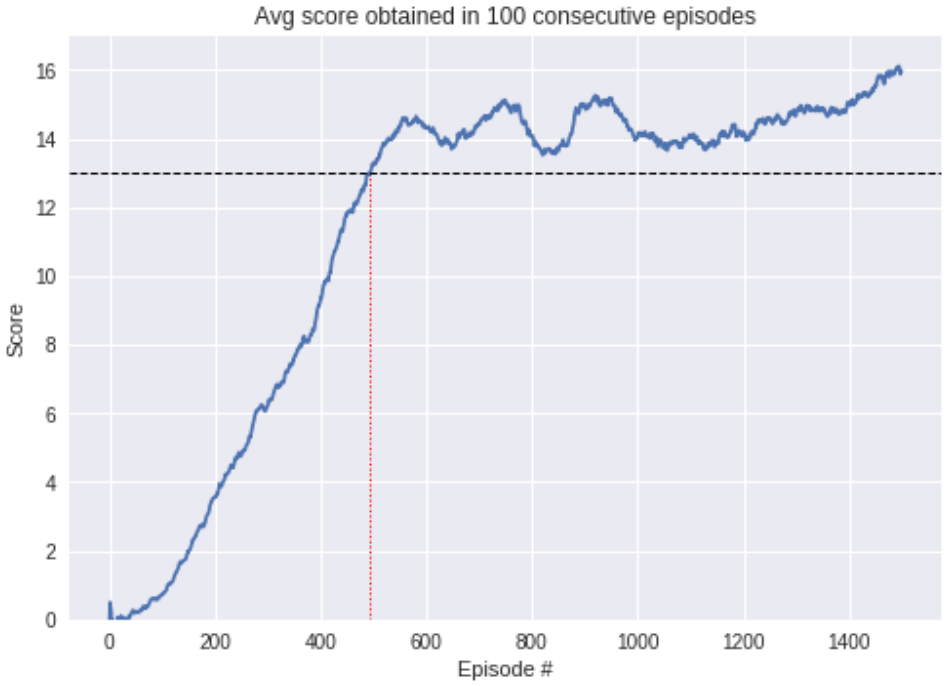




DDQN_c5000

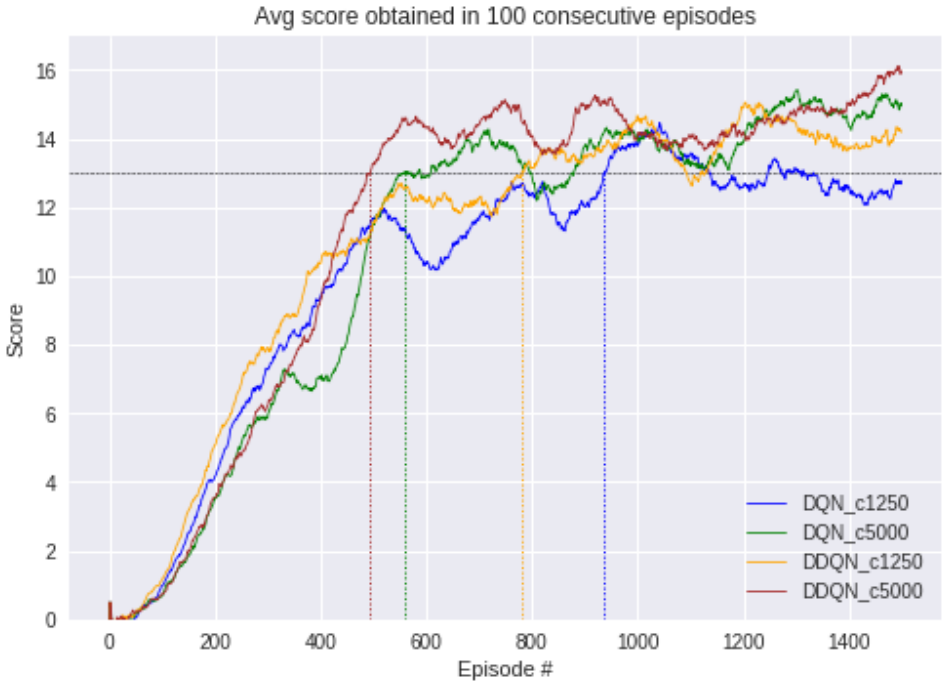
Problem solved in 494 episodes. After being solved, each episode has:

- Avg score: 14.58
- std: 3.95



All this information could be summarized as follows:

Agent	Solved in	Avg score after solved	Std score after solved
DQN_c1250	937	13.01	3.93
DQN_c5000	557	13.99	4.07
DDQN_c1250	782	14.0	4.08
DDQN_c5000	494	14.58	3.95



It could be seen that to use a very small C lead to worse results, so it could be inferred that the correlation between the target and the local network is not completely minimized. Double DQN implementation lead to a better performance, although all the approaches have too much noise in the rewards obtained through the training. This may well be because the experience tuples used for training are randomly sampled from a big replay buffer, where some of the samples could not be the best for the learning process.

Ideas for Future Work

In the future, it would be interesting to implement other well-known approaches such as Dueling Networks and Prioritized Experience Replay, being the last one the one that might improve more current performance as the agent would be trained with the most relevant experience tuples.

Moreover, Policy-based methods might work well too, so in the future it would be interesting to test them too.

Acknowledgement

The developed code is based on the codebase provided in the lessons of the Deep Reinforcement Learning Nanodegree of Udacity.

Bibliography

[1] Human-level control through deep reinforcement learning

[2] Deep Reinforcement Learning with Double Q-learning